Optimizing Explicit Hydrodynamics for Power, Energy, and Performance

E. A. Leon, I. Karlin, R. E. Grant

April 23, 2014

International Conference on Cluster Computing
Chicago, IL, United States
September 8, 2015 through September 11, 2015

Optimizing Explicit Hydrodynamics for Power, Energy, and Performance

Edgar A. León Ian Karlin
Livermore Computing
Lawrence Livermore National Laboratory
Livermore, California, USA
{leon,karlin1}@llnl.gov

Ryan E. Grant
Center for Computing Research
Sandia National Laboratories
Albuquerque, New Mexico, USA
regrant@sandia.gov

***Abstract*—Practical considerations for future supercomputer designs will impose limits on both instantaneous power consumption and total energy consumption. Working within these constraints while providing the maximum possible performance, application developers will need to optimize their code for speed alongside power and energy concerns. This paper analyzes the effectiveness of several code optimizations including loop fusion, data structure transformations, and global allocations. A per component measurement and analysis of different architectures is performed, enabling the examination of code optimizations on different compute subsystems. Using an explicit hydrodynamics proxy application from the U.S. Department of Energy, LULESH, we show how code optimizations impact different computational phases of the simulation. This provides insight for simulation developers into the best optimizations to use during particular simulation compute phases when optimizing code for future supercomputing platforms. We examine and contrast both x86 and Blue Gene architectures with respect to these optimizations.**

***Keywords*-optimization; power; energy; performance;**

I. Introduction

For Exascale class supercomputers, power and energy will become first-class operating concerns. Potential instantaneous power caps as well as operational cost concerns over total energy usage will lead system and application developers to pursue optimizations for power and energy consumption, in addition to performance. The implications of power and energy concerns for supercomputers have a broader impact than compliance with the U.S. Department of Energy 20 MW system power limit. Many practical issues arise when using tens of MWs of power, such as large and fast swings in power and load prediction to match power generation and distribution to anticipated needs. This is an important issue to address for utility companies providing power to data centers as high power demands or large swings in power demand can increase the cost of energy delivery. Therefore, the management of power instead of energy can be more important than energy alone, especially if energy/power conservation goals are cost related. These factors reinforce the need to understand how applications can be optimized for both performance and power on future systems.

When analyzing the performance and power implications of different application optimizations, it is important to examine the impact of individual regions, in addition to the entirety of the application. Runtime-based power or energy saving approaches have shown that benefits can be achieved without application code changes by examining an entire execution period and finding energy and power friendly ways to execute that application [1]. However, such approaches are limited to aggregate static approaches or runtime methods requiring resources during execution that may introduce noise. In addition, because energy saving runtime methods do not alter application code, no benefits can be derived from increasing computational or memory efficiency. Code optimization at a per-region basis offers a fine-grained environment where developers can optimize their code for performance and potential power savings. By analyzing an application at the same level of granularity as a developer, we can provide insight to them on how performance tuning, power usage, and energy costs interplay. Runtime methods, even those that attempt to take advantage of phases, can only react to the existing code's behavior during the current execution phase. The methods we use in this study provide insight on a per-phase basis and, for non-global optimizations, a better understanding into which optimizations work well for individual kernels. This enables much finer grained optimizations and subsequently broadens opportunities to reduce power and save energy compared to a whole-application based approach.

This paper analyzes the impact of different optimizations at a sub-application phase for three different architectures: IBM Blue Gene/Q (BG/Q), server-class Intel Ivy Bridge, and consumer-class AMD Fusion APU. This comparison builds on our previous work [2], where we analyzed the impact of different optimization methods at an application level granularity for a BG/Q system. By comparing optimization techniques on a low power HPC-specific architecture, an enterprise class server system, and a low-power, low-cost consumer APU based system, this paper provides insight into optimization techniques for multiple potential Exascale architectural approaches. Exploring multiple architectures is key to determining what optimization methods may be broadly applicable and which might be of great use on only a subset of architectures.

This paper characterizes the individual kernels in an explicit hydrodynamics application, identifying the most important kernels in terms of execution time, power and energy consumption for multiple different architectures. We investigate the impact of a group of optimization techniques on each phase and provide insight into the optimization methods that are beneficial across many architectures versus those that are important for only certain architectures. We outline our contributions as follows.

1. This work is the first of its kind in providing a per application phase, per system component analysis of the power and energy impacts of program transformations.

2. Our results from an explicit hydrodynamics proxy application apply to other codes with similar computational kernels including the Lawrence Livermore National Laboratory (LLNL) applications Ares and ALE3D [3], [4].

3. We identify significant correlations between program optimizations and power, energy, performance, and machine characteristics.

4. We provide guidance to system and application developers when tuning for performance, power, and energy.

5. We demonstrate that the analysis of applications on a per physics package, phase, or region is necessary to fully realize the benefits of optimizations. Similarly, fine-grain, component-based analysis is necessary to leverage potential improvements in power and energy consumption.

II. LULESH

The Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (LULESH) mini-app was originally developed as one of the five challenge problems for the DARPA UHPC program. Explicit hydrodynamics can consume up to one third of the compute cycles at the U.S. Department of Defense data centers. LULESH provides a simplified source code that contains the data access patterns and computational characteristics of larger hydrodynamics codes. It uses an unstructured hexahedral mesh with two centerings and solves the Sedov problem. Because of its smaller size, LULESH allows for easier and faster performance tuning experiments on various architectures. The successful lessons learned from it can then be applied back to larger production codes [3]. LULESH has been ported to a wide variety of programming models to explore their various performance and productivity advantages [5] and is currently being used in the Department of Energy's Extreme-Scale Technology Acceleration program[1], the CORAL procurement[2], and the ExMatEx co-design center[3].

A. Program Optimizations

This paper focuses on how optimizations impact performance alongside power and energy, therefore, we study optimizations that have been shown to decrease execution time [3]. These optimizations include loop fusion, data layout transformations, global allocation, and vectorization. Some of the lessons learned from these optimizations on LULESH have been ported into full hydrodynamics applications such as ALE3D [3], a structural engineering code, and Ares [4], a high energy density physics code. In this work, we focus on loop fusion, data layout transformations, and global allocation optimizations because they reduce data motion, which will be increasingly important on future machines in terms of power and performance.

[1]https://asc.llnl.gov/fastforward/

[2]https://asc.llnl.gov/CORAL-benchmarks/

[3]http://codesign.lanl.gov/projects/exmatex/index.html

Loop fusion is an optimization that combines multiple loops with the same iteration space together. When loops that access the same arrays are combined, the amount of data needed to move through the memory hierarchy is reduced [6]. An example of this optimization for two loops in LULESH is shown below. The velocity and position update is on the left and the fused version on the right.

```
for (i=0; i<nodes; ++i)      for (i=0; i<nodes; ++i)
  // Calc. new velocity        // Calc. new velocity
  xdtmp=xd[i]+xdd[i]*dt        xdtmp=xd[i]+xdd[i]*dt
  if (FABS(xdtmp)<ucut)        if (FABS(xdtmp)<ucut)
    tmp = Real_t(0.0)            tmp = Real_t(0.0)
  xd[i] = xdtmp                xd[i] = xdtmp
for (i=0; i<nodes; ++i)
  // Calc. new position        // Calc. new position
  x[i] += xd[i] * dt           x[i] += xd[i] * dt
```

The version of LULESH we use for our fused code contains 12 loops from the original 45. Fusing loops further would result in transformations that are not possible or difficult to maintain in the full codes modeled by LULESH [3].

Data layout transformations involve changing a *struct of arrays* to an *array of structs*. These transformations can reduce the amount of data moved through the memory hierarchy by combining accesses to indirectly accessed data structures [7]. Also, they can reduce the number of streams being prefetched from memory, resulting in more effective use of hardware stream prefetchers. In LULESH, we combine arrays into 10 different structures, see Table I. The listing below shows the resulting data structure, *coords* (right side), from the combination of the nodal position variables *x*, *y*, and *z* (left side).

```
  double x[n];          struct xyz { double x,y,z; }
  double y[n];          coords xyz[n];
  double z[n];
```

Global allocation involves moving all the *malloc* and *free* statements outside of the timestep loop. Therefore, all temporary variables are allocated once and then reused without freeing space throughout the program. The listing below provides an example of the global allocation transformation; original code on the left, resulting code on the right.

```
while(time < maxtime)          malloc( temp_space )
  malloc( temp_x )             while(time < maxtime)
  for(i=0; i<nodes; ++i)         for(i=0; i<nodes; ++i)
    // calculations                // calculations
  free( temp_x )                 for(i=0; i<elems; ++i)
  malloc( temp_y )                 // calculations
  for(i=0; i<elems; ++i)       free( temp_space )
    // calculations
  free( temp_y )
```

Another option that can result in a similar performance gain is using a thread aware allocation library such as *TCMalloc*[4] In some cases these libraries can result in the same performance as global allocation without the programming challenges or memory costs needed to maintain global temporary variables.

It is worth noting that recent attempts to provide automatic compilation-based code optimizations [8] similar to the ones

[4]http://goog-perftools.sourceforge.net/doc/tcmalloc.html

we study here have been difficult, working on simplified versions of LULESH that limit its applicability to a full code. In this work we do not modify the unstructured mesh access and while we limit the techniques we can use, we know they are applicable to the code being represented by LULESH.

TABLE I. TRANSFORMED DATA STRUCTURES.

Description	Arrays	Description	Arrays
Temporary Forces	fx_elem, fy_elem, fz_elem	Coordinates	x, y, z
Principle Strains	dxx, dyy, dzz	Velocities	xd, yd, zd
Accelerations	xdd, ydd, zdd	Forces	fx, fy, fz
Velocity Gradient	delv_xi, delv_eta, delv_zeta	Presure and Q	p, q
Coordinate Gradient	delx_xi, delx_eta, delx_zeta	Q Terms	ql, qq

III. MACHINE ARCHITECTURE AND MEASUREMENT INFRASTRUCTURE

Our experiments consist of measuring the impact of code optimizations on power, energy, and execution time for three different architectures: IBM BG/Q, Intel Ivy Bridge, and AMD Fusion (see Table II). In the remainder of this section, we describe the infrastructures we used to measure power, energy, and performance events.

TABLE II. MACHINE ARCHITECTURE.

	Node Processor(s)	Memory	OS
IBM BG/Q	PowerPC A2, 16 user cores, SMT-4, 1.6 GHz	DDR3-1333 MHz, 16 GB, 43 GB/s	CNK
Intel Ivy Bridge	Xeon E5-2695V2 dual socket, 10x2 cores, SMT-2, 2.4 GHz	DDR3-1866 MHz, 128 GB, 59.7x2 GB/s	SLES11
AMD Piledriver	AMD A10-5800K APU, 4 CPU cores, 3.8 GHz	DDR3-1600 MHz, 16 GB, 12.8 GB/s	RHEL6

On BG/Q, we used IBM's high-resolution environmental monitoring, capable of measuring power and energy at a node-board granularity (32 compute nodes). Each of the two Direct Current Assemblies (DCAs) included on a board has a microprocessor unit that measures current and voltage of at most seven domains (see Table III). Through the EMON2 (Environmental Monitoring version 2) API, a user application retrieves cumulative energy consumption at a given point in time. With two snapshots, the EMON2 library computes the energy difference and the average power consumption for the given interval. In our BG/Q experiments, we capture power and energy consumption every 10 ms for all seven domains. More information on BG/Q's high-resolution power infrastructure can be found elsewhere [9]. We measure performance events along-side power snapshots through the BGQT library [9], which is built on top of the EMON2 API and the Blue Gene Hardware Performance Monitoring (BGPM) API.

On the Ivy Bridge and Fusion systems, we used Penguin's PowerInsight measurement device, a small out-of-band instrument designed to capture power/energy measurement readings from individual nodes and provide these results to a centralized master node [10]. It includes an ARM Cortex A8 processor with 256MB of RAM and a variety of output methods such as a 10/100 Ethernet connector, which was used in this work for collecting the power measurements. The main ARM

TABLE III. BG/Q AND POWERINSIGHT POWER DOMAINS.

BG/Q	**PowerInsight**	**Alias**
BQC core logic power	CPU socket power	Core
SDRAM-DDR3, BQC DDR3 I/O	DDR3-1600	Memory
Optical module power Optical module power, PCI Express BQC and BQL HSS I/O BQC core array power BQL core power	Motherboard & Chipset Solid State Drive Miscellaneous PCIe InfiniBand NIC (PCIe)	Other

Cortex board is connected to a sensor daughter board which enables the connection of multiple inline sensors, allowing per component measurements to be taken at high speed. The measurement accuracy of PowerInsight has been determined to average 1.8% of the true values. More information on the PowerInsight design and its verification can be found elsewhere [10]. The PowerInsight measurement was set to 10 ms sampling intervals for all of the available sampling domains (see Table III), identical to the BG/Q sampling period.

IV. EXPERIMENTAL METHODOLOGY

To study individual optimizations and their combined effect on power, energy, and performance, we developed 2^3 versions of LULESH representing all optimization combinations (see Table IV). We compare the different optimizations on each architecture relative to an unoptimized version (NoOpt) of LULESH. We use the term unoptimized to mean no fusion, global allocation, data layout transformations, or their combinations as opposed to compiler optimization level 0. On BG/Q, NoOpt was built with the IBM compiler while on Fusion and Ivy Bridge, we used the Intel compiler. Note that we employ the best compiler available for each platform.

TABLE IV. LULESH OPTIMIZATIONS.

ac	Global allocation
dl	Data structure layout transformations
fu	Loop fusion
BG/Q NoOpt	IBM xlc++ with -O3 -qhot -qstrict -qsmp=omp
x86 NoOpt	Intel icc with -O3 -openmp

For each version of LULESH, we ran one process per NUMA domain (e.g. socket) with the process pinned to that socket. We ran this way to mimic an MPI process of a large application as it would be run in production, where each MPI task would use a single NUMA domain. On BG/Q, we ran 1 process with 60 OpenMP threads (15 cores using 4 threads per core) per node and replicated this workload across the entire node-board. We dedicated the 16th core of one node for the power and profiling monitoring thread. On the Ivy Bridge we ran 1 process per NUMA domain with a total of 20 OpenMP threads (one per physical core) per node. Although Hyperthreading was enabled, only one OpenMP thread per core was used. On the Fusion we ran 1 process with 4 OpenMP threads per node to fill all four cores; the GPU available on the Fusion APU was not used. All platforms used a problem size of 120^3 per node, ran for 130 iterations, and started measurements after the application's initialization phase.

To understand the static and dynamic power consumption of LULESH, we executed a single thread benchmark that performs no work on each of the systems. The power consumed by this benchmark represents the static power. All of our systems were configured to not enter sleep states for CPU cores, so this is a measure of power/energy of no-active-work performance with a small amount of potential OS noise. To calculate dynamic power we subtracted this static power measurement from the total power consumed by LULESH.

In this paper we report average power rather than peak power for a few reasons. For a programmer seeking to maximize performance, power over a period of time is what will put thermal load on a chip and cause it to reduce its frequency or allow it to increase into Turbo mode. For a full application where load imbalance occurs, being able to save power on lightly loaded nodes and allow other nodes with heavier loads to run at higher clock speeds may be more important than preventing spikes on the most loaded nodes. In addition, if data centers' users are charged for electricity, average power times runtime and total energy usage are equivalent and these are the two free variables that can be traded off when optimizing.

V. Power, Energy, and Performance at a Glance

In order to evaluate the power and energy impact that the proposed optimizations have on a per region basis on LULESH, it is desirable to investigate multiple computer architectures to identify trends. As the optimization of applications will impact dynamic power consumption, it is helpful to quantify the dynamic versus static power consumption of architectures. Figure 1 shows the static and dynamic power draw for multiple components for a BG/Q system and two x86 architectures. We observe that BG/Q has a significant static power component, comprising 75% of the overall power draw. The x86 systems have 22% and 36% static power draws for the Ivy Bridge and Fusion systems respectively. This illustrates two key points. First, that BG/Q, while well known for its low power architecture, leaves limited opportunities for power savings due to its large static component. Second, that the core and memory power draw dominates (78% or more) overall system power, which for the x86 architectures is also predominantly dynamic power.

Figure 1 shows the dynamic power drawn by each architecture. The two low power architectures, BG/Q and AMD Fusion, both draw similar amounts of power per processing node. However, the Fusion architecture has a much larger dynamic power draw than the BG/Q, likely because it has fewer cores but at a higher clock rate (4-3.8GHz cores vs. 16+1-1.6GHz cores). The comparison with the Ivy Bridge system in terms of core power draw should be made carefully, the Ivy Bridge system is a dual socket system, so the core power draw of 139 Watts is an aggregate of two 10-core CPUs. Examining the memory power consumption, the Fusion and BG/Q systems have very similar power consumption, which is expected given their identical amount of memory (16 GB), with the small difference between them corresponding to the memory frequency differences of the DDR3-1333 in the BG/Q versus the DDR3-1600 memory in the Fusion system. The Ivy Bridge system appears to draw significantly more power for its memory, but this corresponds to 8x as much memory (128 GB vs. 16 GB), running at 1866 MHz.

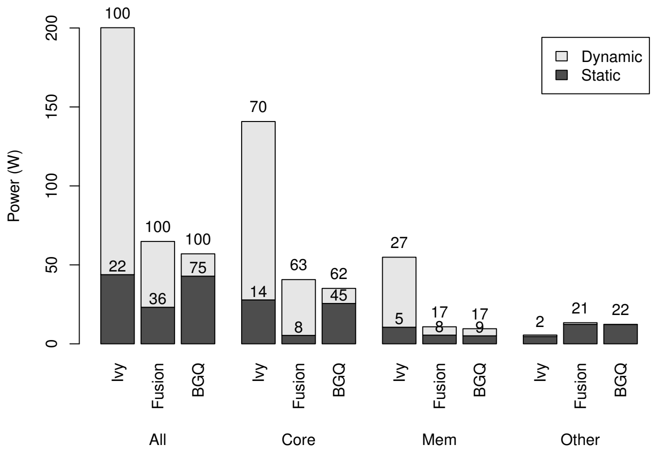

Fig. 1. Power consumed by the NoOpt configuration itemized by component for all three architectures.

Figure 2 shows the normalized runtime and dynamic power and energy consumption for the various optimized versions of LULESH. The BG/Q and Ivy Bridge systems benefit from fused loops, global allocations and data layout transformations. The AMD Fusion system, with its consumer-grade APU does not see this same trend. The best configuration for the Fusion APU is fused loops with global allocations. It should be noted that the Fusion system has the greatest improvement in runtime of any of the systems, due to the ability of the optimizations to work around system bottlenecks. The Fusion system has significantly smaller caches (2MB vs 25MB and 32MB for the Ivy Bridge and BG/Q respectively), and lower memory bandwidth, which creates opportunities for improving performance significantly through better memory management.

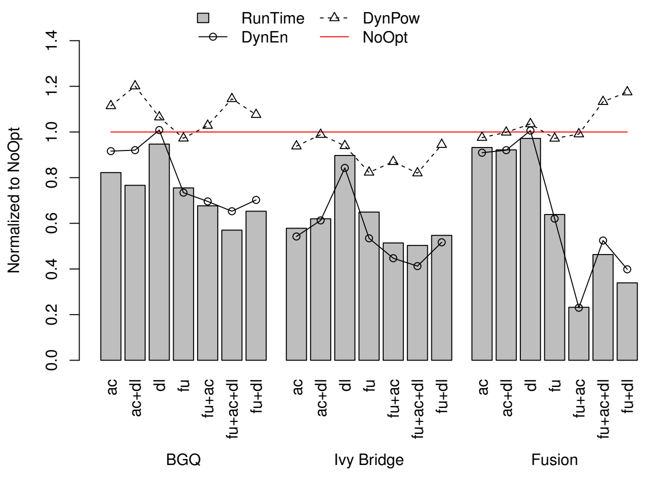

Fig. 2. Execution time, dynamic energy, and dynamic power for each optimization on BG/Q, Ivy Bridge and Fusion

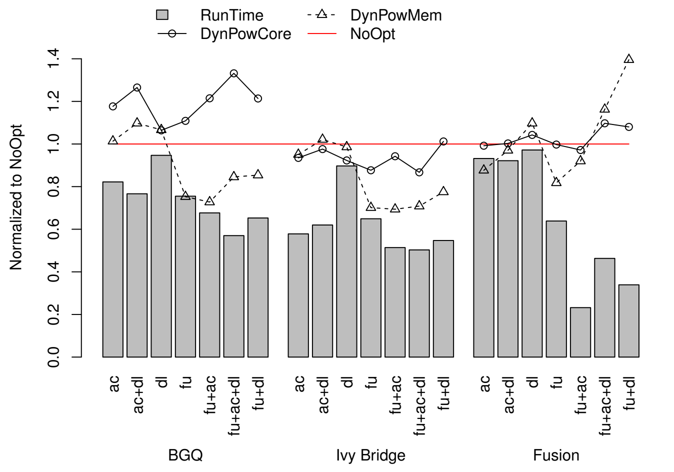

Fig. 3. Execution time and dynamic power for each power domain and optimization on BG/Q, Ivy Bridge, and Fusion.

For each of the x86 architectures it is interesting to note that the best optimizations in terms of runtime are also relatively good for instantaneous power consumption, being approximately equal to or better than the baseline case with no optimizations. Combined with decreased runtime, this translates into significant energy savings for both the Ivy Bridge and Fusion systems. This is an important finding showing that the performance of the application can be improved with no major increase in power consumption. Unfortunately these results also show that significant reductions in instantaneous power consumption at an application level are difficult to achieve on most platforms with the application optimizations that were explored. The BG/Q case differs from the x86 results slightly. For the BG/Q system, the third best runtime set of optimizations draws power approximately equal to the non-optimized case. The best optimization combination for runtime and energy consumes approximately 15% more power than the baseline case. This difference is primarily due to core power consumption differences between the BG/Q optimizations as seen in Figure 3. The x86 Fusion system has relatively flat CPU power consumption across the optimization with lower memory power draw for non data layout optimization combinations. The Ivy Bridge system shows slightly higher core power consumption with lower memory power consumption, which balances out to be neutral in terms of the overall system.

While we find that global allocations alone work well for the two systems with the largest caches, the Fusion system did not see significant benefit from using them. The Fusion system also saw a significant benefit from loop fusion combined with global allocations. The impact was the largest of any of the architectures with any combination of optimizations. This illustrates that from a high level point of view, some optimizations can be applied regardless of the architecture used and see benefits. However, other optimizations can yield very high returns on some platforms while providing much more modest improvement on others. Finally, we observe that some combinations of optimizations are not beneficial on certain systems (e.g. fu+ac+dl relative to fu+ac on the Fusion system, see Figure 3). The increase in power consumption for the Fusion's memory subsystem can be traced to the combination of two optimizations, loop fusion and data layout. Such optimizations when combined together increase the likelihood of the Fusion's memory controller recognizing a limited number of simultaneous streams (as data layout decreases the total number of streams), which is closer to the maximum of 8 streams supported by the controller. In addition, the loop fusion also allows the memory controller to recognize certain patterns more easily, leading to many more pre-fetch operations than would occur in the non-optimized case.

Our analysis so far focused on the single-node versions of LULESH as opposed to the MPI multi-node versions. The lessons from the former, however, apply to the latter for the following reasons: (1) the LULESH regions we analyzed, which account for over 90% of the overall runtime, do not perform network communication, therefore the compute benefits at the node level should carry over to multi-node systems, assuming no significant load-imbalance; (2) LULESH is not communication intensive (less than 10% of runtime is spent in communication for large runs) and is load balanced; and (3) while network power is of interest, there is little opportunity for saving power with optimizations because the majority of power is consumed by network SERDES and does not change based on usage as illustrated in the ”other” column in Figure 1 for AMD and BG/Q (Intel results do not include network power).

To demonstrate that, indeed, the lessons learned from our experiments carry over the MPI multi-node case, we ported fusion and global allocation (fu and ac) to the MPI version of LULESH, ran these versions on 64 nodes on 2 different architectures, and compared them with the single-node versions. Figure 4 shows the effect of optimizations on execution time, dynamic power and energy, and memory and processor power. On BG/Q, the single-node energy, power, and performance match accurately the MPI data (BGQ-MPI) differing by no more than 3%. On AMD, the single-node data and the MPI data (Fusion-MPI) show similar behavior but the margin of difference is greater (up to 9% in memory power), because the small caches on the AMD cores are susceptible to evictions caused by the MPI library and OS noise.

VI. Fine-Grain Analysis of Optimizations

In the previous section we compared and contrasted the program-level effects of optimizations on power, performance, and energy across three architectures. We now investigate the effects of optimizations at a finer-granularity by focusing on five code regions that consume over 90% of LULESH's execution time. The execution time of each region on BG/Q using the NoOpt configuration is shown in Table V. Only results from BG/Q are presented because the percentage of total time spent in each region is within 5% for the other architectures.

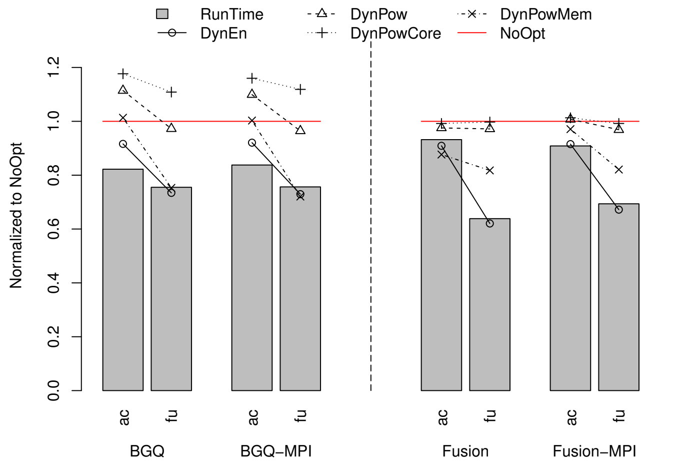

Fig. 4. Single-node vs. MPI versions of LULESH for BG/Q (left) and Fusion (right). BG/Q should be compared against BG/Q-MPI and Fusion against Fusion-MPI.

TABLE V. EXECUTION TIME PER REGION ON BG/Q.

	Region 1	Region 2	Region 3	Region 4	Region 5
Runtime (secs)	11.85	32.02	1.10	13.80	16.21
Fraction of total	15.80%	42.71%	1.46%	18.41%	21.62%

A. LULESH: A Look Inside

In LULESH each region we selected represents a part of the production code that often has multiple options associated with it or connects regions of code that have multiple options. For example, the EOS and MaterialApply in Region 5 will be different depending on what material model is being used and Region 2 for the hourglass filter will frequently change based on the problem being run. Due to the need to modularly swap out the piece of physics based on the problem being run or during runtime as conditions change in the simulation, optimizations across these regions, such as loop fusion, are often impractical to maintain in a production code because of the combinatorial number of versions needed to support each physics option combination. The five regions we look at were chosen because they present a diverse set of properties, some of which change significantly when the code is optimized, and represent about 90% of the application's runtime. It is worth noting that when loop fusion is applied, each region becomes a single loop. In this section, we briefly describe each region and highlight what makes them unique.

Region 1 involves the *stress* calculation routines. It starts with a memory-bound initialization routine followed by a compute-bound calculation of element volumes and normal forces followed by a memory-bound update of nodal values. **Region 2** performs the *hourglass* calculation. It performs a memory-intense copy of data followed by a series of compute-bound small matrix-matrix multiplications followed by a memory-bound update of nodal force values. **Region 3** consists of two memory-bound loops that update the velocity and position of mesh nodes. **Region 4** includes *CalcKinematicsForElems* and *CalcMonotonicQForElems*, which gather values from nodes to element centers followed by compute-intense calculations. **Region 5** includes *MonotonicQforRegions* and *MaterialApply*, which have a significant number of control flow instructions and instructions with dependencies. It also includes the routine *EvalEOSForElems*, which performs a significant number of memory copies and control flow operations.

B. Characterizing Region-based Performance

In this section, we focus on the performance characteristics of each region and the impact of optimizations on them. For brevity, we focus on the BG/Q architecture and leave the cross-architecture analysis to the next section. Table VI shows the performance characteristics of all five regions on BG/Q. We show both the unoptimized case (NoOpt) and the best program-level optimization in terms of execution time (Best Time, i.e., fu+ac+dl). On this machine the maximum theoretical IPC is two, split evenly between integer and floating point instructions and STREAM memory bandwidth peaks at 28.5 GB/s.

TABLE VI. CHARACTERISTICS OF LULESH'S REGIONS.

	BG/Q	INT%	FPU%	IPC	L1 Hits%	MemBW (GB/s)
NoOpt	Region 1	47.4	52.6	0.541	90.4	18.38
	Region 2	62.5	37.5	0.554	93.6	15.56
	Region 3	74.1	25.9	0.216	75.7	20.88
	Region 4	45.5	54.5	0.654	88.4	9.12
	Region 5	62.4	37.6	0.321	77.2	13.72
Best Time	Region 1	42.7	57.3	0.791	93.9	19.02
	Region 2	60.1	39.9	0.924	96.8	10.33
	Region 3	78.5	21.5	0.248	85.9	21.40
	Region 4	46.4	53.6	0.878	96.7	9.19
	Region 5	50.3	49.7	0.606	90.6	7.81

INT = load/store/integer instructions; FPU = floating-point unit instructions; IPC = instructions per cycle.

We observe in Table VI that Regions 1 and 3 both use a significant fraction of the available memory bandwidth before and after the best optimization. While Region 3 is limited by memory bandwidth and has a low IPC in both the optimized and unoptimized case, Region 1 has a moderate IPC before optimization and a higher one after. Therefore, Region 1 has both memory and compute intense characteristics after optimization. For Region 4, its compute intensity increases with optimization resulting in a compute-bound region. In general, IPC and cache hit rates increase as a result of optimizations. For these regions, other metrics are not affected significantly.

Regions 2 and 5 have lower bandwidth requirements and higher IPC after optimization. These changes are due to eliminating extra data motion required for temporary data structures (ac) and the corresponding increase in compute intensity from fused loops (fu). Region 2 changes from a combination of compute and memory intensive loops to compute-bound after optimizations. Region 5, on the other hand, has a significant increase in the number of floating point operations relative to integer operations, a much higher L1 cache hit rate, and a relatively low IPC for its bandwidth requirements. Therefore, Region 5 is dominated by instruction pipeline latency due to many dependent instructions and branches when tuned.

Figure 5 provides a more detailed view of how the optimizations impact performance on BG/Q. Each optimization affects the performance of regions differently. Global allocation (ac) has a significant impact on Regions 1 and 2. While this is the primary optimization that improves performance of these regions, fusion (fu) also has a significant impact. Data layout (dl) also has a small but positive impact on both regions.

Fusion and data layouts both decrease the performance of Region 3 individually, but improve its performance when combined. On BG/Q this is because fusing loops overwhelms the limited number of stream prefetchers (four per thread), but when combined with data layout transformations the number of streams is reduced. Global allocation helps in some optimization combinations and hinders performance in others.

Region 4 gets a small performance boost from loop fusion, but almost all of its performance gain is from data layout transformations. The performance of Region 5 improves when allocation is applied by itself or with data layouts, however, once loop fusion is applied allocation has no effect since loop fusion contracts all the temporary arrays in Region 5 to scalars.

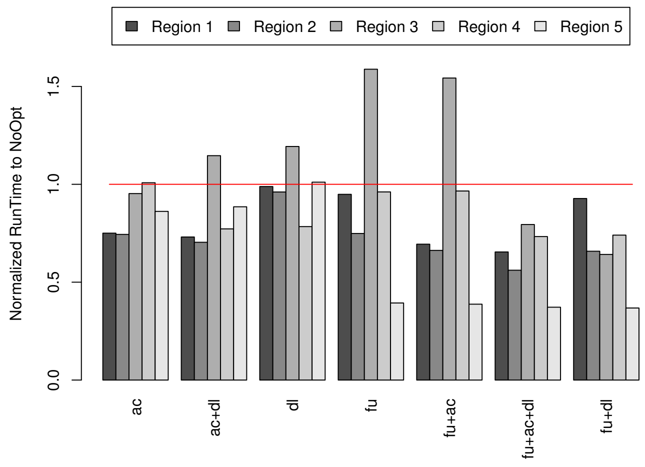

Fig. 5. The effect of optimizations on execution time broken down by region on BG/Q. Execution time is normalized to NoOpt.

C. In-depth Analysis and Cross-architectural Implications

In this section, we analyze in detail the behavior of the top two Regions–2 and 5–in terms of execution time. We start with a correlation between the improvements observed in runtime and low-level architectural counters, which we will refer to later in this section. Table VII shows the results of the correlation analysis for both Pearson's, which look for linear relationships, and Spearman's, which examine monotonic relationships. We observe that the correlations are very strong for all of the Pearson's correlations except for Region 5 on the Fusion, where the results are only strongly correlated. For the Spearman's correlations (using an α of 0.1), all of the values pass the null hypothesis test, showing that a correlation exists.

Figure 6 shows execution time and dynamic power for Region 2. This compute intense region presents some cross-architectural trends. In all cases the lowest memory power and highest core power is found in optimizations with loop fusion (fu). Applying loop fusion increases the compute intensity of the loops and results in more computation per unit time, but less data motion. Except for two cases on Fusion (fu+ac+dl and fu+dl), loop fusion always improves memory power and execution time on all architectures. Table VII substantiates these results. The BG/Q is best correlated to its instruction throughput, while the x86 architectures' performance is correlated to memory throughput. The optimizations to this code region have reduced the data movement requirements and have shifted the code balance on the BG/Q from being memory bound to compute bound. The x86 architectures have much stronger integer compute units and are still bounded by memory performance after the optimizations. The very high correlations for both the BG/Q and Ivy bridge show that they are both strongly bound by compute and memory, respectively. The Fusion core shows a good but not outstanding correlation to memory performance indicating that with further data movement optimizations, it may become compute bound like the BG/Q.

TABLE VII. CORRELATIONS OF PERFORMANCE TO MONITORING COUNTERS FOR A SUBSET OF LULESH'S REGIONS.

	System	Best Counter	Pearson Corr.	Spearman Corr.
Region 2	BGQ	Inst. Exec.	0.974	0.905
	Ivy Bridge	L2 Misses	0.969	0.952
	Fusion	L1 Misses	0.752	0.809
Region 5	BGQ	L2 Misses	0.999	0.976
	Ivy Bridge	L1 Misses	0.775	0.809
	Fusion	L2 Hits	0.618	0.714

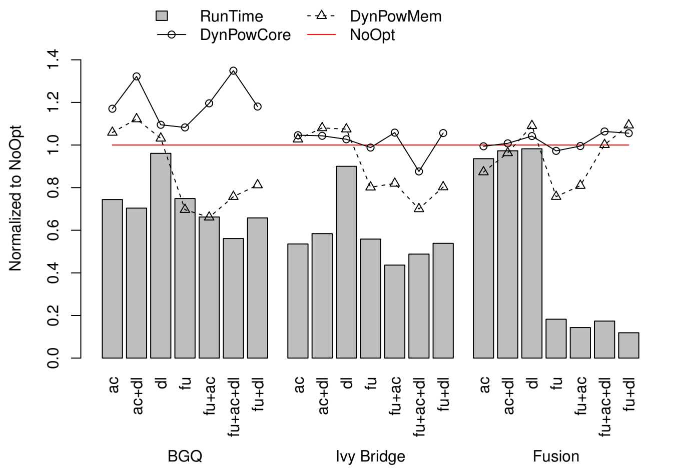

Fig. 6. The effect of optimizations on execution time, dynamic core power, and dynamic memory power for Region 2 across all architectures.

The differences shown in the performance correlations of the optimizations used in two different regions of interest for LULESH are illustrative of the difficulties in performing optimizations that are beneficial across many different architectures. For regions that can be well optimized, the factors limiting performance/energy of the code may be different between architectures or may switch after optimization for some architectures and not others. For example, applying further optimizations for data movement on code that may be compute limited on a given architecture is unlikely to yield positive results and may even be counter productive. However, other architectures may benefit from further data movement optimizations, leading to situations in which enhancing performance/energy efficiency on one architecture leads to declining performance/energy efficiency on another.

Region 5 has many memory-bound loops when loop fusion is not applied. As Figure 7 shows, loop fusion is necessary to extract the best performance on all systems. On BG/Q and Ivy Bridge, loop fusion needs to be combined with data layout transformations (fu+dl) and on Fusion with allocation (fu+ac). In most cases though just applying loop fusion results in the best or nearly the best memory power. However, unlike the other regions where optimizing for execution time does not result in a poor choice for power, Region 5 shows a tradeoff. On BG/Q and Ivy Bridge just applying fusion is best for power, but adding data layout transformations improves performance at a steep power cost. On the Fusion system only applying allocation is best for power, but not much better than adding in loop fusion, which results in a large performance gain.

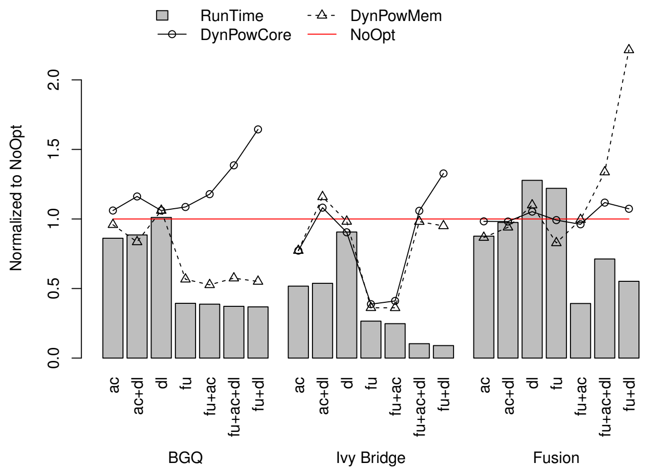

Fig. 7. The effect of optimizations on execution time, dynamic core power, and dynamic memory power for Region 5 across all architectures.

It is clear from examining the performance counter correlations in Table VII that the optimizations for Region 5 have not changed the (memory) boundedness of the code for any of the architectures. Both x86 architectures correlate well to memory based counters, while the BG/Q has an extraordinarily high correlation to L2 performance. This emphasizes the extreme dependency on memory of this region on the BG/Q, while showing that other architectures are also memory bound. This helps to illustrate the across-the-board benefit to memory/data movement based optimizations on this region.

D. Dynamic Energy Analysis

Examining the dynamic energy consumption over the five regions of LUELSH, the most interesting region for study is Region 4, which is shown in Figure 8. We leave the other regions out because in those there is a one-to-one correlation between the best optimization for overall dynamic energy usage and the best optimization for performance. Often, but not always, that correlation extends to both dynamic memory energy and dynamic core energy. However, the code version in Region 4 with the best dynamic energy usage is different on both Ivy Bridge and BG/Q than the best performing in execution time (see Figure 8).

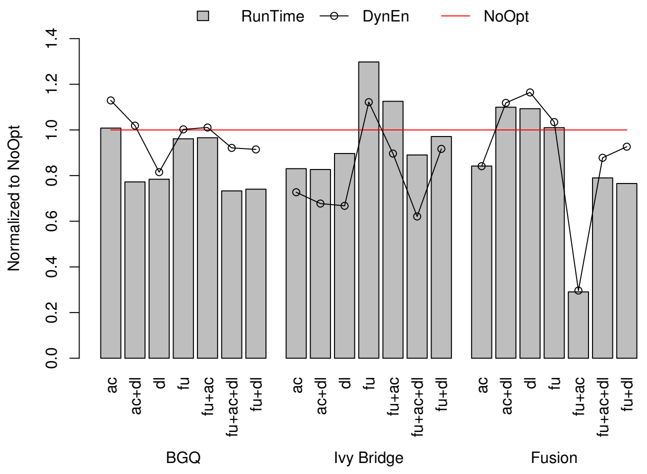

Fig. 8. The effect of optimizations on execution time and dynamic energy for Region 4 across all architectures.

VII. Discussion

Table VIII presents a high level overview of the best optimizations for dynamic power, dynamic power for each subsystem, dynamic energy, and execution time of the application. In this section, we discuss how overall program results can obscure fine-grain performance and power differences between architectures and code regions. While common in performance analysis this is less common in power studies. We discuss whether applying different optimizations to different regions has the potential to improve power or performance of an entire application. For this analysis it is important to note that loop fusion (fu) can be applied independently to each region; allocation (ac) can be applied independently to each variable, but if a variable is used multiple times there will be a memory usage cost; and data layout transformations (dl) must be applied globally.

A. Architecture Specific Lessons

For BG/Q the overall (program as a whole) fastest set of optimizations (fu+ac+dl) is also the best set for three of the five regions and is the second best set for the other regions. Thus, there is a clear best choice for performance. When optimizing for core power, not applying any optimizations is best if a global whole-program optimization scheme is used. However, this contrasts with optimizing for memory power where applying allocation and loop fusion is best. However, overall power is best when applying just loop fusion.

TABLE VIII. BEST OPTIMIZATIONS FOR DYNAMIC POWER AND ENERGY AND RUNTIME ACROSS REGIONS AND FOR THE WHOLE PROGRAM.

		Region 1	**Region 2**	**Region 3**	**Region 4**	**Region 5**	***Program***
BG/Q	PowCore	NoOpt	NoOpt	fu+ac	dl	NoOpt	*NoOpt*
	PowMem	fu+dl	fu+ac	dl	fu	fu+ac	*fu+ac*
	Power	fu+dl	fu	fu	NoOpt	fu	*fu*
	Energy	fu+ac+dl	fu+ac+dl	fu+dl	dl	fu	*fu+ac+dl*
	RunTime	fu+ac+dl	fu+ac+dl	fu+dl	fu+ac+dl	fu+dl	*fu+ac+dl*
Ivy	PowCore	dl	fu+ac+dl	NoOpt	dl	fu	*NoOpt*
	PowMem	fu	fu+ac+dl	fu+dl	fu+ac+dl	fu+ac	*fu+dl*
	Power	fu	fu+ac+dl	fu	fu+ac+dl	fu	*fu+ac+dl*
	Energy	ac	fu+ac+dl	fu	fu+ac+dl	fu+ac	*fu+ac+dl*
	RunTime	ac	fu+ac	fu+ac+dl	ac+dl	fu+dl	*fu+ac+dl*
Fusion	PowCore	fu+ac	fu	ac+dl	fu+ac	fu+ac	*fu+ac*
	PowMem	fu+ac	fu	fu+dl	fu	fu	*fu*
	Power	fu+ac	fu	ac+dl	ac	ac	*fu*
	Energy	fu+ac	fu+dl	ac+dl	fu+ac	fu+ac	*fu+ac*
	RunTime	fu+ac	fu+dl	dl	fu+ac	fu+ac	*fu+ac*

A high level view of system power, however, obscures that it is better not to optimize Region 4 when minimizing total power. Also, some regions and domains benefit from data layout changes or allocation in addition to loop fusion. Further complicating the power optimization story is that often the optimal optimization for either power or runtime is only slightly better than the next one to three optimization combinations. Therefore, finding a globally optimal solution, by only selectively applying allocation, data layouts, and loop fusion to various regions and variables will be challenging even for the small number of optimizations and code regions we consider in this paper.

For the Fusion system the results are less complicated. Loop fusion and allocation demonstrate the best power and performance tradeoff. These two optimizations result in the best performance and core power, and the third best memory and overall power. Also these optimizations are best, or close to the best for performance and core power, for all regions except for Region 3. Only if memory power becomes a larger fraction of overall machine power would not applying allocation be considered. Region 3 presents a stark contrast, with data layouts being important for core power and performance optimization. The Region 3 results show that for a code with more memory-bound routines, optimizing for this architecture might be significantly different than our LULESH result (Region 3 accounts for a small percentage of LULESH runtime).

On the Ivy Bridge system, the fastest and most power efficient optimizations at a program level hide significant differences between power domains and code regions. The optimization (fu+ac+dl) that results in the best power usage for Regions 2 and 4 and the best runtime for Region 3 are the same as the optimizations with the best overall power and runtime. However, this program-level view obscures that Regions 1 and 5 would consume less power if only loop fusion was applied. In particular, Region 5's runtime reduction from optimization is significant enough to reduce its contribution to overall power usage. If instead of power we analyze dynamic energy use, then the version of code with all optimizations applied would be best and second best respectively on these metrics. Therefore, without a power-bound the best optimizations converge, while under a power bound tuning on a per-region basis becomes attractive. Therefore, the Ivy Bridge system gives an example, where the quantity being optimized for, how the machine is being run and the balance of properties of code at a fine-level can significantly impact how to best tune the code.

B. Cross Architecture Lessons

The results and analysis in this paper show that there are some optimizations that are generally beneficial across architectures, while other optimizations are beneficial only to a subset of the architectures. This is not surprising due to the large differences between the systems. The BG/Q processor has a low clock frequency (1.6 GHz), in-order execution and a large, but power efficient L2 cache. The AMD Fusion architecture has much higher frequency (3.8 GHz) and uses out-of-order execution on fewer (4), more sophisticated cores than BG/Q, and has smaller but higher energy caches. Finally, the Ivy Bridge system has ten sophisticated cores with a frequency of 2.4 GHz and a high power L3 cache that is the largest per core of the group.

Table VIII shows that loop fusion is often part of the suite of optimizations that is best for power and runtime for all architectures across most power domains and regions. Also, allocation, in general helps performance across all regions. Therefore, an important take away from this paper is the insight that there are some optimizations that are generally useful, and some that can best be deployed on a specific basis. The Fusion results show that architecture specific optimizations for power on a per region basis, can make the extra work to develop and maintain architecture specific code versions justified.

Data layout transformations are sometimes important to performance and power as seen in Region 3 for Fusion and Region 4 for Ivy Bridge. Also, on BG/Q for Region 3 data layout transformations along with loop fusion significantly improves performance. However, the overall impact of data layouts was small, isolated, and often negative, with significant whole program performance gains only occurring on BG/Q. Also, data layouts did not significantly reduce power on any architecture. However, their positive impact in Region 3 shows that they might be important for applications that have many memory-bound regions and should be explored further in that context.

A surprising result from this study is that optimizations that reduce data motion often reduced core power. While not applicable to BG/Q, loop fusion and data layouts reduced core power for many regions despite increasing IPC. We believe this reduction is due to less data motion within the on-chip memory hierarchy, but cannot demonstrate this with current power measurement technology (cannot measure individual cache draw). Therefore, a takeaway from this work is that power efficient cache technology can lead to significant power savings. In addition, if fine-grain power optimizations are important on future machines, then a similar finer-grain power measurement infrastructure would be necessary.

Another significant cross-architecture result is that while performance trends, and to a lesser extent power trends, are similar at the program optimization level for BG/Q and Ivy Bridge, this is not as evident at a region level. In addition, the Fusion power and performance trends are significantly different than the other two machines. These results show that tuning differently for power at a fine-grain on each architecture is profitable.

C. Applying Transformations to Production Codes

Some of the optimizations explored in this paper have been applied back to full production hydrodynamics applications to improve performance. For example, loop fusion transformations found in Regions 1 and 2 were applied to xALE, a subset of ALE3D, and were later applied to the full code [3]. Allocation transformations were also applied, through the TCMalloc library, to Ares, another hydrodynamics code [4]. In this case the transformations are only profitable on BG/Q. The main impediment to applying data layout transformations to production codes is their invasive nature and that different architectures need different layouts [7].

Therefore, the application of the transformations in this paper to optimize for power are in some cases applicable to current production hydrodynamics codes. While today these transformations have been focused only on performance as machines become more power constrained the knowledge presented in this paper provides a basis for trading power and performance when optimizing a code. In other cases, transformations such as data layouts will become applicable as programmers adopt programming models, e.g., Kokkos [11] or Chapel [12], that allow the data layout to be decoupled from the array access patterns in loops.

VIII. Related work

Recent related work can be classified into the following areas: modeling power and energy in HPC; and analyzing performance, power, and energy trade offs. Modeling is a useful vehicle to approximate power, energy, and temperature on systems that lack direct measurement capabilities. These models employ performance information to estimate the desired parameters [13], [14]. In addition, modeling is essential to predict power and energy on future systems. A few examples include modeling to estimate leakage energy on a cache hierarchy [15], modeling power and energy at finer granularities in terms of space and time [16], and modeling individual system components [17], [18].

The second main body of work related to this paper explores tradeoffs between power, energy, and execution time, especially for compiler transformations. Wang et al. [8] used a polyhedral optimizer to generate multiple program variants with different optimizations including loop fusion, unrolling, tiling, and vectorization. Their results using LULESH show a strong correlation between execution time and energy consumption. However, they only analyzed a single parallel region within LULESH and to do so they had to remove the indirect access patterns from LULESH. A single variant of the most expensive parallel region took hours to compile, making cross-architecture portability and significant search space exploration impractical. In addition, since their state-of-the-art tool does not attempt to fuse loops across multiple regions and cannot handle code with indirection, their lessons learned are not portable back into the application code as we have shown [3]. Wang showed the power of auto-tuning techniques, but these approaches are limited in their ability to consume production applications because of the code complexities. This is one reason why in this paper we focus on manual optimization rather than automatic optimization. We believe both are appropriate, but applicable in different situations.

Balaprakash et al. [19] presents a multi-objective optimization framework to understand the trade-offs of performance, power, and energy. This formulation of objective functions can be used in auto-tuning environments with competing objectives. The authors demonstrated empirically that these trade offs exist. Tiwari et al. [13] uses compiler optimizations (loop tiling and unrolling) to validate their power and energy models of processor and memory components. Deshpande et al. [15] showed that certain compiler optimizations have a small effect on cache leakage.

Our work leverages existing power monitoring capabilities on BG/Q [9] and, similar to previous work, identifies the performance events that capture the driving force behind power and energy consumption on LULESH. This work also leverages the power monitoring capabilities on the x86 systems by using the PowerInsight measurement devices [10].

This related work serves as a basis for *analyzing* why certain program transformations trade off power and energy, which is the focus of our work. Our contributions include a detailed analysis of power and energy consumption of several optimizations that have been successfully applied to explicit hydrodynamic codes in terms of performance on multiple architectures. Unlike previous work, we provide insights that explain why optimizations that can be applied to production codes today trade off power, energy and performance at a per-region application basis. This fine-grained analysis over multiple architectures shows optimizations that are generally useful and those that are more targeted to specific architectures, or code regions. This work is the first of its kind to explore fine-grained analysis coupled with accurate high-frequency power sampling on a variety of architectures and is important in providing insight to application developers interested in preparing their code bases for next generation supercomputers that will, for the first time, have to operate in power/energy conscious and potentially power constrained environments.

IX. Summary

This paper investigates the effects of code optimizations on LULESH on a per-region basis for two subsystems (core and memory) on three different architectures. By analyzing how optimizations change the power, performance, and energy draw of an application, we affirm that tuning for these objectives at a sub-application phase is necessary to fully realize their potential. By performing our analysis at the granularity an application developer performs optimizations, the insights we gained in this paper are transferable directly back to application developers and enables them to reason about which optimizations to apply based on code characteristics, their computer system, and the relative power consumption by various domains within their system. We summarize our findings as follows:

General observations

1. Our results extend beyond LULESH. We have successfully applied our findings to full hydrodynamics applications, including Ares and ALE3D, resulting in significant improvements in time-to-solution.
2. Whole-program, single-domain power and performance analysis can limit program optimizations' gains in energy efficiency. Power and energy profiles are phase and domain dependent.
3. Current power measurement technology is limited to coarse-grain domains that hinder correlations between performance and power, e.g., individual cache power draw within a processor is not directly measurable.

Cross-architecture lessons

4. Optimizing for performance and energy are strongly correlated, i.e., performance optimized code results in energy optimized code.
5. In many cases, optimizations do not increase power significantly. As such, they can be applied under a similar power budget.
6. The effect of optimizations on power is significantly larger on the memory system than the processor. This emphasizes the relevance of optimizations on future systems, which will increase the ratio of memory power to total power.

Architecture-specific findings

7. Optimizations result in higher processor power on BG/Q than x86 machines relative to each architecture's baseline.
8. On Fusion, fu+ac is generally a good choice to improve performance and power.
9. On BG/Q and Ivy Bridge, although fu+ac+dl results in significant performance and energy improvements, the best optimization combination depends on the objective function and system subcomponent.

Future work includes evaluating these and other (e.g., vectorization) optimizations under different P-states and memory frequencies on a per-region granularity to improve power and energy with a marginal performance impact.

Acknowledgments

We would like to thank Bert Still for his guidance and suggestions as well as Matthew Dosanjh for helping to gather PMC data on the x86 platforms. Prepared by LLNL under Contract DE-AC52-07NA27344. LLNL-CONF-653578. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] J. H. Laros III, K. T. Pedretti, S. M. Kelly, W. Shu, and C. T. Vaughan, "Energy based performance tuning for large scale high performance computing systems," in *Symposium on High Performance Computing*, ser. HPC'12, Orlando, FL, Mar. 2012.

[2] E. A. León and I. Karlin, "Characterizing the impact of program optimizations on power and energy for explicit hydrodynamics," in *Workshop on High-Performance, Power-Aware Computing*, ser. HP-PAC'14. Phoenix, AZ: IEEE, May 2014.

[3] I. Karlin, J. McGraw, E. Gallardo, J. Keasler, E. A. León, and B. Still, "Memory and parallelism exploration using the LULESH proxy application," in *International Conference for High Performance Computing, Networking, Storage and Analysis; Poster*, ser. SC'12. IEEE, 2012.

[4] I. Karlin and M. Collette, "Strong scaling bottleneck identification and mitigation in Ares," in *Nuclear Explosive Code Development Conference Proceedings (NECDC14)*, January 2015.

[5] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *IEEE International Parallel & Distributed Processing Symposium*, 2013.

[6] G. Gao, R. Olson, V. Sarkar, and R. Thekkath, "Collective loop fusion for array contraction." in *Workshop on Languages and Compilers for Parallel Computing*, Aug. 2004.

[7] K. Sharma, I. Karlin, J. Keasler, J. R. McGraw, and V. Sarkar, "User-specified and automatic data layout selection for portable performance," Rice University, Tech. Rep. TR13-03, April 2013.

[8] W. Wang, J. Cavazos, and A. Porterfield, "Energy auto-tuning using the polyhedral approach," in *Workshop on Polyhedral Compilation Techniques*, ser. IMPACT'14, Jan. 2014.

[9] R. Bertran, Y. Sugawara, H. M. Jacobson, A. Buyuktosunoglu, and P. Bose, "Application-level power and performance characterization and optimization on IBM Blue Gene/Q systems," *IBM Journal of Research and Development*, vol. 57, no. 1/2, 2013.

[10] J. H. Laros, P. Pokorny, and D. DeBonis, "Powerinsight-a commodity power measurement capability," in *International Green Computing Conference*, ser. IGCC'13. IEEE, 2013, pp. 1–6.

[11] H. C. Edwards, D. Sunderland, V. Porter, C. Amsler, and S. Mish, "Manycore performance-portability: Kokkos multidimensional array library," *Scientific Programming*, vol. 20, no. 2, pp. 89–114, 2012.

[12] B. L. Chamberlain, S.-E. Choi, D. Steven J, and A. Navarro, "User-defined parallel zippered iterators in Chapel," in *Conference on Partitioned Global Address Space Programming Models*, Galveston, TX, October 2011.

[13] A. Tiwari, M. A. Laurenzano, L. Carrington, and A. Snavely, "Modeling power and energy usage of HPC kernels," in *Workshop on High-Performance, Power-Aware Computing*. IEEE, May 2012.

[14] S. L. Song, K. Barker, and D. Kerbyson, "Unified performance and power modeling of scientific workloads," in *International Workshop on Energy Efficient Supercomputing*. ACM, 2013.

[15] A. Deshpande and J. Draper, "Leakage energy estimates for HPC applications," in *International Workshop on Energy Efficient Supercomputing*, ser. E2SC'13. ACM, Nov. 2013.

[16] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, "Enabling accurate power profiling of HPC applications on exascale systems," in *International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, Jun. 2013.

[17] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *International Symposium on Microarchitecture*, ser. MICRO-36. IEEE, Dec. 2003.

[18] R. Bertran, M. Gonzalez, X. Martorell, N. Navarro, and E. Ayguade, "Decomposable and responsive power models for multicore processors using performance counters," in *International Conference on Supercomputing*. ACM, 2010.

[19] P. Balaprakash, A. Tiwari, and S. M. Wild, "Multi objective optimization of HPC kernels for performance, power, and energy," in *International Workshop on Energy Efficient Supercomputing*, ser. E2SC'13. Denver, CO: ACM, Nov. 2013.